Laboratorium 4.

0) Przygotowanie środowiska

W konsoli/terminalu wpisujemy kolejno

```
$ cd
$ mkdir haskell-lab4
$ cd haskell-lab4

# sprawdzamy dostępność narzędzia stack
$ which stack
$ stack --version
```

1) Typy w Haskellu: *type vs. newtype*

1. W konsoli GHCi wpisujemy kolejno

```
ghci> type t1 = Int
ghci> type T1 = Int
ghci> type T2 = (T1,Double)
ghci> type T3 = (a,a)
ghci> type T4 a = (a,a,a)
ghci> type T5 a = [a]
ghci> type T6 a b = [([a],[b])]
ghci> type T7 a b = a -> b
ghci> type T8 a b = a -> (a -> b) -> Int
ghci> type T9 a = (a, T9 a)
ghci> type T10 = (Int, T10)
```

2. W pliku ex1.hs wpisujemy

```
polarToCartesian :: Floating a => (a,a) -> (a,a)
polarToCartesian (r,phi) = (r * cos phi, r * sin phi)
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `polarToCartesian`

```
ghci> :t polarToCartesian
ghci> let (x1,y1) = polarToCartesian (1,pi/4)
ghci> let (x2,y2) = polarToCartesian (x1,y1) -- !!!
ghci> polarToCartesian . polarToCartesian $ (1,pi/4) -- !!!
```

3. W pliku ex1.hs dodajemy

```
type CartesianCoord' a = (a,a)
type PolarCoord' a = (a,a)
```

```
polarToCartesian' :: Floating a => PolarCoord' a -> CartesianCoord' a
polarToCartesian' (r,phi) = (r * cos phi, r * sin phi)
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `polarToCartesian'`

```
ghci> :t polarToCartesian'
ghci> let (x1,y1) = polarToCartesian' (1,pi/4)
ghci> let (x2,y2) = polarToCartesian' (x1,y1) -- !!!
ghci> polarToCartesian' . polarToCartesian' $ (1,pi/4) -- !!!
```

Co zyskaliśmy wprowadzając aliasy/synonimy typów?

4. W pliku ex1.hs dodajemy

```
newtype CartesianCoord'' a = MkCartesianCoord'' (a,a)
newtype PolarCoord'' a = MkPolarCoord'' (a,a)

polarToCartesian'' :: Floating a => PolarCoord'' a -> CartesianCoord'' a
polarToCartesian'' (MkPolarCoord'' (r,phi)) = MkCartesianCoord'' (r * cos phi, r * s
```

5. ***Zadania***:
    1. Sprawdzić działanie funkcji `polarToCartesian''` dla poprawnych i niepoprawnych danych wejściowych; rozważyć zalety i wady powyższych trzech implementacji
    2. (*opcjonalne*) Napisać odpowiedniki powyższych typów i funkcji dla układów: cylindrycznego i sferycznego
    3. (*opcjonalne*) Sprawdzić działanie funkcji

    ```
    personInfoToString :: (String,String,String) -> String
    personInfoToString (nm,snm,addr) =
     "name: " ++ nm ++ ", surname: " ++ snm ++ ", addr: " ++ addr
    ```

    np. dla następujących danych wejściowych

    ```
    ghci> personInfoToString ("Michail", "Berlioz", "ul. Sadowa 302a, m.50")
    ghci> personInfoToString ("ul. Sadowa 302a, m.50", "Stiopa", "Lichodiejew")
    ```

    4. (*opcjonalne*) Dodać aliasy

    ```
    type Name' = String
    type Surname' = String
    type Address' = String
    type PersonInfo' = (Name', Surname', Address')
    type PersonInfoToStringType' = PersonInfo' -> String
    ```

oraz napisać i sprawdzić działanie funkcji

```
personInfoToString' :: PersonInfoToStringType'
```

5. (*opcjonalne*) Wykorzystując `newtype` napisać funkcję

```
personInfoToString'' :: PersonInfo'' -> String
personInfoToString'' ... =
```

która będzie bardziej ‘odporna’ na błędne dane wejściowe; sprawdzić działanie funkcji

2) Algebraiczne typy danych 1: product & sum types, record syntax

1. W pliku ex2.hs wpisujemy

```
-- product type example (one constructor)
data CartInt2DVec = MkCartInt2DVec Int Int -- konwencja: prefix 'Mk' dla konstruktor

xCoord :: CartInt2DVec -> Int
xCoord (MkCartInt2DVec x _) = x

yCoord :: CartInt2DVec -> Int
yCoord (MkCartInt2DVec _ y) = y
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli GHCi wpisujemy kolejno

```
ghci> :i CartInt2DVec
ghci> :t CartInt2DVec
ghci> :t MkCartInt2DVec -- analizujemy typ konstruktora

ghci> let p12 = CartInt2DVec 1 2
ghci> let p12 = MkCartInt2DVec 1 2

ghci> :t xCoord -- analizujemy typ xCoord
ghci> xCoord p12
ghci> :t yCoord -- analizujemy typ yCoord
ghci> yCoord p12
ghci> xCoord $ MkCartInt2DVec 5 10
```

Co dałoby wprowadzenie w deklaracji `CartInt2DVec` aliasów/synonimów

```
type X = Int
type Y = Int
```

2. W pliku ex2.hs dodajemy

```
data Cart2DVec' a = MkCart2DVec' a a
```

```
xCoord' :: Cart2DVec' a -> a
xCoord' (MkCart2DVec' x _) = x

yCoord' :: Cart2DVec' a -> a
yCoord' (MkCart2DVec' _ y) = y
```

(zapisujemy zmiany, wczytujemy plik do GHCi `:r`); w konsoli GHCi wpisujemy kolejno

```
ghci> :i Cart2DVec'
ghci> :t MkCart2DVec'

ghci> :t MkCart2DVec' 1 2
ghci> :t MkCart2DVec' 1.0 2.0
ghci> :t MkCart2DVec' 1.0 2
ghci> :t xCoord' $ MkCart2DVec' 5 10
ghci> yCoord' $ MkCart2DVec' 5.0 10.0
```

3. W pliku ex2.hs dodajemy

```
data Cart2DVec'' a = MkCart2DVec'' {x::a, y::a}

xCoord'' :: Cart2DVec'' a -> a
xCoord'' (MkCart2DVec'' {x = xVal, y = _}) = xVal

yCoord'' :: Cart2DVec'' a -> a
yCoord'' (MkCart2DVec'' {y = yVal, x = _}) = yVal -- uwaga na kolejność x,y
```

(zapisujemy zmiany, wczytujemy plik do GHCi `:r`); w konsoli GHCi wpisujemy kolejno

```
ghci> :i Cart2DVec''
ghci> :t MkCart2DVec''
ghci> :t xCoord''
ghci> :t yCoord''
ghci> :t x -- dlaczego ta funkcja istnieje (skoro jej nie deklarowaliśmy/definiowali
ghci> :t y -- jw.?

ghci> let p23 = MkCart2DVec'' {x = 2, y = 3}
ghci> xCoord'' p23
ghci> x p23
ghci> yCoord'' p23
ghci> y p23

ghci> xCoord'' $ MkCart2DVec'' {x=1, y=2}
ghci> xCoord'' $ MkCart2DVec'' 1 2
ghci> yCoord'' $ MkCart2DVec'' {x=1, y=2}
ghci> yCoord'' $ MkCart2DVec'' 1 2
```

4. W pliku ex2.hs usuwamy deklaracje/definicje `xCoord''` i `yCoord''` pozostawiając tylko

```
data Cart2DVec'' a = MkCart2DVec'' {x::a, y::a}
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli GHCi wpisujemy kolejno

```
ghci> :i Cart2DVec''
ghci> :t MkCart2DVec''
ghci> :t x
ghci> :t y

ghci> let p23 = MkCart2DVec'' {x = 2, y = 3}
ghci> x p23
ghci> y p23
```

5. W pliku ex2.hs dodajemy

```
-- sum type example (two constructors)
data List a = EmptyL | Cons a (List a) deriving Show

head' :: List a -> a
head' EmptyL      = error "head': the empty list has no head!"
head' (Cons x xs) = x
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli GHCi wpisujemy kolejno

```
ghci> head' EmptyL
ghci> head' Cons 1
ghci> head' (Cons 1 EmptyL)
ghci> head' $ Cons 1 EmptyL
ghci> Cons 1 EmptyL -- show in action :)
ghci> head' $ Cons 1 $ Cons 2 EmptyL
```

6. W pliku ex2.hs dodajemy

```
-- enum type example (special case of sum type)
data ThreeColors = Blue |
                   White |
                   Red

type ActorName = String

leadingActor :: ThreeColors -> ActorName
leadingActor Blue  = "Juliette Binoche"
leadingActor White = "Zbigniew Zamachowski"
leadingActor Red   = "Irene Jacob"
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `leadingActor`

7. ***Zadania***:

    1. Zdefiniować funkcje dostępowe np. `xCoord3D` itd. dla następującego typu (reprezentacja wektorów w 3D, współrzędne kartezjańskie)

    ```
    {-
    uwaga: ta sama nazwa* dla:
     - konstruktora typu (po lewej)
     - konstruktora danych/wartości (po prawej)

     * druga (obok omówionej poprzednio -- z prefiksem 'Mk') powszechna konwencja w
    -}
    Cart3DVec a = Cart3DVec a a a
    ```

    a następnie sprawdzić ich działanie

    2. Wykorzystując *record syntax* napisać nową wersję `Cart3DVec`, a następnie sprawdzić istnienie odpowiednich (wygenerowanych przez kompilator) funkcji dostępowych
    3. (*opcjonalne*) Zdefiniować odpowiednie wersje typów (bez i z *record syntax*) dla wektorów 2D w układzie biegunowym
    4. (*opcjonalne*) Dla obu powyższych typów zdefiniować funkcję `polarToCartesian`
    5. (*opcjonalne*) Napisać odpowiedniki powyższych typów i funkcji dla układów: cylindrycznego i sferycznego
    6. Zdefiniować funkcję

    ```
    area :: Shape -> Float
    ```

    obliczającą pole powierzchni (figury płaskiej); założyć, że

    ```
    data Shape = Circle Float |
                 Rectangle Float Float
    ```

    7. (*opcjonalne*) Zdefiniować funkcję

    ```
    rootValue :: Tree a -> a
    ```

    zwracającą element przechowywany w korzeniu drzewa binarnego zdefiniowanego w następujący sposób

    ```
    data Tree a = EmptyT |
                  Node a (Tree a) (Tree a)
                  deriving Show
    ```

    Uwaga: dla pustego drzewa funkcja powinna zgłaszać błąd (funkcja `error`)
    8. Zdefiniować typ wyliczeniowy `TrafficLights` dla sygnalizacji świetlnej oraz funkcję

    ```
    actionFor :: TrafficLights -> String
    ```

podającą, co powinien robić kierowca, widząc dane światło
9. (*opcjonalne*) Zdefiniować typ wyliczeniowy dla ‘akcji’ z poprzedniego zadania (działanie kierowcy w danej sytuacji)

3) Algebraiczne typy danych 2: rekursja strukturalna

1. W pliku ex3.hs wpisujemy

```
data BinIntTree = EmptyIntBT |
                  IntNodeBT Int BinIntTree BinIntTree

sumBinIntTree :: BinIntTree -> Int
sumBinIntTree EmptyIntBT = 0
sumBinIntTree (IntNodeBT n lt rt) = n + sumBinIntTree lt + sumBinIntTree rt
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `sumBinIntTree`, np.

```
ghci> sumBinIntTree EmptyIntBT
ghci> sumBinIntTree $ IntNodeBT 1 EmptyIntBT EmptyIntBT
ghci> sumBinIntTree $ IntNodeBT 1 (IntNodeBT 2 EmptyIntBT EmptyIntBT) (IntNodeBT 3 E
```

2. W pliku ex3.hs dodajemy (uogólniamy typ)

```
data BinTree a = EmptyBT |
                 NodeBT a (BinTree a) (BinTree a)

sumBinTree :: (Num a) => BinTree a -> a
sumBinTree EmptyBT = 0
sumBinTree (NodeBT n lt rt) = n + sumBinTree lt + sumBinTree rt
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `sumBinTree`, np.

```
ghci> sumBinTree EmptyBT
ghci> sumBinTree $ NodeBT 1 EmptyBT EmptyBT
ghci> sumBinTree (NodeBT 1 (NodeBT 2 EmptyBT EmptyBT) (NodeBT 3 EmptyBT EmptyBT))
```

3. W pliku ex3.hs dodajemy

```
data Expr a = Lit a | -- literal/value a, e.g. Lit 2 = 2
              Add (Expr a) (Expr a)

eval :: Num a => Expr a -> a
eval (Lit n) = n
eval (Add e1 e2) = eval e1 + eval e2

show' :: Show a => Expr a -> String
```

```
show' (Lit n) = show n
show' (Add e1 e2) = "(" ++ show' e1 ++ "+" ++ show' e2 ++ ")"
```

(zapisujemy zmiany, wczytujemy plik do GHCi) i sprawdzamy działanie funkcji `show'` i `eval`, np.

```
ghci> show' (Lit 2)
ghci> show' (Add (Lit 1) (Lit 2))
ghci> eval (Lit 1)
ghci> eval (Add (Lit 1) (Lit 2))
```

4. ***Zadania***:

    1. Napisać definicje następujących funkcji:

    ```
    depthOfBT :: BinTree a -> Int -- głębokość drzewa binarnego
    flattenBT :: BinTree a -> [a]  -- napisać trzy wersje: preorder, inorder, postc
    mapBT :: (a -> b) -> BinTree a -> BinTree b -- funkcja map dla drzewa binarnegc
    insert :: Ord a => a -> BinTree a -> BinTree a -- insert element into BinTree
    list2BST :: Ord a => [a] -> BinTree a -- list to Binary Search Tree (BST)
    ```

    2. (*opcjonalne*) Napisać definicje następujących funkcji:

    ```
    occurs :: Eq a => a -> BinTree a -> Int -- liczba wystąpień elementu w drzewie
    elemOf :: Eq a => a -> BinTree a -> Bool -- sprawdzenie, czy element znajduje s
    reflect :: BinTree a -> BinTree a -- 'odbicie lustrzane' drzewa binarnego
    minElemOf :: Ord a => BinTree a -> a
    maxElemOf :: Ord a => BinTree a -> a
    foldBinTree :: (a -> b -> b -> b) -> b -> BinTree a -> b -- fold dla drzewa bir
    ```

    3. (*opcjonalne*) Zdefiniować `mapBT` przy pomocy `foldBinTree`
    4. (*opcjonalne*) Zaproponować implementacje funkcji `zipBT` (`zip` dla drzew binarnych)
    5. (*opcjonalne*) Uogólnić typ `BinTree` do

    ```
    data GTree a = Leaf a |
                   GNode [GTree a]
                   deriving Show
    ```

    oraz napisać odpowiednie wersje następujących funkcji:

    ```
    sumGTree :: Num a => GTree a -> a
    elemOfGTree :: Eq a => a -> GTree a -> Bool
    depthOfGTree :: GTree a -> Int
    mapGTree :: (a -> b) -> GTree a -> GTree b
    ```

```
flattenGTree :: GTree a -> [a]
countGTreeLeaves :: GTree a -> Int
```

6. Rozszerzyć definicję typu `data Expr a` o operacje odejmowania i mnożenia oraz zmodyfikować odpowiednio funkcje `show'` i `eval`
7. (*opcjonalne*) Zmodyfikować definicję `data Expr a` wykorzystując 'infixowe konstruktory' (*uwaga*: pierwszym znakiem nazwy musi być w tej notacji ':')

```
data Expr a = Lit a |
              Expr a :+: Expr a |
              Expr a :-: Expr a |
              ...
```

oraz odpowiednio zmienić funkcje `eval` i `show'`

8. (*opcjonalne*) Rozszerzyć możliwości 'kalkulatora' o operacje logiczne, np. w następujący sposób:

```
data Expr a = Lit a |
              Op Ops (Expr a) (Expr a) |
              If (BExpr a) (Expr a) (Expr a)

data Ops = Add | Sub | Mul

data BExpr a = BoolLit Bool |
               And (BExpr a) (BExpr a) |
               Or (BExpr a) (BExpr a) |
               Not (BExpr a) |
               Equal (Expr a) (Expr a) |
               Greater (Expr a) (Expr a)

eval :: Num a => Expr a -> a
eval (Lit n) = n
eval (Op Add e1 e2) = eval e1 + eval e2
...

bEval :: BExpr a -> Bool
bEval (BoolLit b) = b
...
```

4) Typy wyższego rzędu (*ćwiczenie opcjonalne*)

1. W konsoli GHCi wpisujemy kolejno (i analizujemy wyniki):

```
ghci> data A0 = MkA0 Int
ghci> :t MkA0 -- typ konstruktora danych (wartości)
ghci> :k A0
```

```
ghci> data A1 a = MkA1 a -- A1 - konstruktor typu
ghci> :t MkA1
ghci> :k A1  -- typ/'kind' konstruktora typu

ghci> data A2Prod a b = MkA2Prod a b
ghci> :t MkA2Prod
ghci> :k A2Prod

ghci> data A2Sum a b = A2SumA a | A2SumAB a b
ghci> :t A2SumA -- typ pierwszego ('lewego') konstruktora danych
ghci> :t A2SumAB -- typ drugiego ('prawego') konstruktora danych
ghci> :k A2Sum

ghci> :k A1 Double
ghci> :k A2Prod Int -- 'partial application' of the type constructor
ghci> :k A2Prod Int Bool
```

2. W konsoli GHCi wpisujemy kolejno (i analizujemy wyniki):

```
ghci> :i Maybe
ghci> :k Maybe
ghci> :k Maybe Int

ghci> :i Either
ghci> :k Either
ghci> :k Either Int
ghci> :k Either Int Double

ghci> :i []
ghci> :k []
ghci> :k [Int]
```

5) Klasy typów i ich instancje 1: dołączanie typu do istniejącej klasy

1. W pliku ex5.hs wpisujemy

```
data MyInt = MkMyInt Int
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```
ghci> MkMyInt 1 == MkMyInt 1
```

2. W pliku ex5.hs dodajemy

```
instance Eq MyInt where
  (==) (MkMyInt i1) (MkMyInt i2) = i1 == i2
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```
ghci> MkMyInt 1 == MkMyInt 1
ghci> MkMyInt 1 == MkMyInt 2
ghci> MkMyInt 1 /= MkMyInt 2 -- dlaczego nie pojawia się błąd?

ghci> MkMyInt 1 < MkMyInt 2
```

3. W pliku ex5.hs dodajemy

```
instance Ord MyInt where
  (<=) (MkMyInt i1) (MkMyInt i2) = i1 <= i2
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```
ghci> MkMyInt 1 <= MkMyInt 2
ghci> MkMyInt 1 < MkMyInt 2 -- dlaczego nie pojawia się błąd?
ghci> MkMyInt 1 > MkMyInt 2 -- jw.?
ghci> MkMyInt 1 >= MkMyInt 2 -- jw.?

ghci> :i Ord -- analizujemy fragment '{-# MINIMAL compare | (<=) #-}'

ghci> MkMyInt 1 + MkMyInt 2
ghci> :i Num -- analizujemy fragment '{-# MINIMAL ... #-}'
```

4. W pliku ex5.hs dodajemy

```
instance Num MyInt where
  (+) (MkMyInt i1) (MkMyInt i2) = MkMyInt (i1 + i2)
  (-) (MkMyInt i1) (MkMyInt i2) = MkMyInt (i1 - i2)
  (*) (MkMyInt i1) (MkMyInt i2) = MkMyInt (i1 * i2)
  negate (MkMyInt i)            = MkMyInt (negate i)
  abs (MkMyInt i)               = MkMyInt (abs i)
  signum (MkMyInt i)            = MkMyInt (signum i)
  fromInteger int               = MkMyInt (fromIntegral int)
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```
ghci> MkMyInt 1 + MkMyInt 2

ghci> MkMyInt 1
ghci> :i Show -- analizujemy fragment '{-# MINIMAL ... #-}'
```

5. W pliku ex5.hs dodajemy

```
instance Show MyInt where
  show (MkMyInt i) = "MkMyInt " ++ show i
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```
ghci> MkMyInt 1 + MkMyInt 2
ghci> MkMyInt 5
ghci> (MkMyInt 2) * (MkMyInt 3 + MkMyInt 4)

ghci> MkMyInt 1 `div` MkMyInt 2
ghci> (MkMyInt 5) ^ 2
```

6. ***Zadania***:
    1. Zmienić definicję

        ```
        data MyInt = MkMyInt Int
        ```

        na

        ```
        newtype MyInt = MkMyInt Int
        ```

        i porównać działanie z poprzednim wariantem (tj. z `data MyInt`)
    2. Dołączyć `BinTree a` do klasy `Eq` (deklaracja `instance Eq a => Eq (BinTree a) where`)
    3. (*opcjonalne*) Dołączyć `Cart3DVec a` do klas `Eq` i `Num`
    4. (*opcjonalne*) Dołączyć typ (reprezentujący liczby wymierne)

        ```
        data Fraction a = Fraction {num::a, denom::a} -- num - numerator, denom - denon
        ```

        do klas `Show`, `Eq`, `Ord` i `Num`; sprawdzić poprawność implementacji
    5. (*opcjonalne*) Zaproponować implementację operatora `<` dla `Cart3DVec a`; sprawdzić, czy kompilator może wygenerować automatycznie (jakąkolwiek) implementację operatora `<`
    6. (*opcjonalne*) Powtórzyć kroki wykonane dla `MyInt` dla typu `data MyDouble`; sprawdzić, instancje których klas kompilator może wygenerować automatycznie (przy pomocy klauzuli `deriving`)
    7. (*opcjonalne*) Sprawdzić działanie operatora `<` dla list, a następnie zdefiniować własną wersję (*wskazówka*: można wykorzystać typ opakowujący stworzony przy pomocy `newtype`)
    8. (*opcjonalne*) Zaproponować kilka wariantów implementacji operatora `<` dla drzew binarnych `BinTree a`; sprawdzić implementację wygenerowaną przez kompilator (przy pomocy klauzuli `deriving`)

6) Klasy typów i ich instancje 2: tworzenie własnych klas typów (*ćwiczenie opcjonalne*)

1. W pliku ex6.hs wpisujemy

```
class Mappable t where
  fMap :: (a -> b) -> t a -> t b
```

```haskell
data Vec3D a = Vec3D {x::a, y::a, z::a} deriving Show

instance Mappable Vec3D where
  fMap f (Vec3D x y z) = Vec3D (f x) (f y) (f z)
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```haskell
ghci> fMap id (Vec3D 1 2 3)
ghci> fMap (+1) (Vec3D 1 2 3)
ghci> fMap (const 1) (Vec3D 1 2 3)
ghci> fMap sqrt (Vec3D 1 2 3)
ghci> fMap abs (Vec3D (-1) (-2) 3)
```

2. W pliku ex6.hs dodajemy

```haskell
newtype Pair a = Pair (a,a) deriving Show

instance Mappable Pair where
  fMap f (Pair (x,y)) = Pair (f x, f y)
```

(zapisujemy zmiany, wczytujemy plik do GHCi); w konsoli wpisujemy

```haskell
ghci> fMap (+1) $ Pair (1,2)
ghci> fMap (length) $ Pair ([1..5], [1..100])
ghci> fMap ($! 3) $ Pair ((^2), (^3))
ghci> fMap ($ 2) $ fMap ((.) (\x -> x + 10)) $ Pair ((^2), (^3))
```

3. ***Zadania***:
    1. Dodać typ

    ```haskell
    data BinTree a = EmptyBT |
                     NodeBT a (BinTree a) (BinTree a)
                     deriving Show
    ```

    do klasy `Mappable` ; sprawdzić działanie `fMap` dla kilku drzew
    2. Dołączyć `Maybe` do klasy `Mappable` (deklaracja `instance Mappable Maybe`)
    3. Dołączyć `Either` do klasy `Mappable` (uwaga na liczbę parametrów typu)
    4. Uzupełnić definicję

    ```haskell
    instance Mappable ((->) a) where
      fMap f g = ___
    ```

    5. Zdefiniować klasę

    ```haskell
    class VectorLike t where
     (|==|) :: Eq a => t a -> t a -> Bool
    ```

```
(|+|), (|-|) :: (Num a) => t a -> t a -> t a
(|*|) :: (Num a) => t a -> t a -> a
(||?), (|-?) :: (Num a, Eq a) => t a -> t a -> Bool -- równoległość i prostopa
vectLength :: Floating a => t a -> a
unitVectOf :: Floating a => t a -> t a
```

a następnie jej instancje dla `Vec2D` i `Vec3D` (tj. dodać te typy do klasy `VectorLike`)

7) Moduły i importy

1. W pliku ex7.hs wpisujemy

```
module Stack (Stack(MkStack), empty, isEmpty, push, top, pop) where

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

newtype Stack a = MkStack [a] deriving Show

empty = MkStack []
isEmpty (MkStack s) = null s
push x (MkStack s) = MkStack (x:s)
top (MkStack s) = head s
pop (MkStack (s:ss)) = (s,MkStack ss)
```

(zapisujemy zmiany, wczytujemy plik do GHCi)

2. W konsoli wpisujemy

```
ghci> :l ex7.hs
ghci> :module -Stack
ghci> import Stack
ghci> :i Stack.[naciskamy TAB]
ghci> :t pop
```

3. W konsoli wpisujemy

```
ghci> :module -Stack
ghci> import Stack ()
ghci> :i Stack.[naciskamy TAB]
ghci> :t pop
```

4. W konsoli wpisujemy

```
ghci> :module -Stack
ghci> import Stack (push, pop)
ghci> :i Stack.[naciskamy TAB]
ghci> :t pop
```

5. W konsoli wpisujemy

```
ghci> :module -Stack
ghci> import qualified Stack
ghci> :i Stack.[naciskamy TAB]
ghci> :t pop
ghci> :t Stack.pop
```

6. ***Zadania***:

    1. Sprawdzić działanie następujących wariantów importu

    ```
    ghci> :module -Stack
    ghci> import qualified Stack (push, pop)

    ghci> :module -Stack
    ghci> import Stack hiding (push, pop)

    ghci> :module -Stack
    ghci> import qualified Stack hiding (push, pop)

    ghci> :module -Stack
    ghci> import Stack as S

    ghci> :module -Stack
    ghci> import Stack as S (push, pop)

    ghci> :module -Stack
    ghci> import qualified Stack as S

    ghci> :module -Stack
    ghci> import qualified Stack as S (push, pop)
    ```

    2. Zmodyfikować pierwszą linię (zapisać zmiany) pliku ex7.hs

    ```
    module Stack (Stack(..), isEmpty, push, pop) where
    ```

    i sprawdzić działanie

    ```
    ghci> :l ex7.hs
    ghci> :module -Stack
    ghci> import Stack
    ghci> :i Stack
    ghci> :t MkStack
    ```

```
ghci> :i push
ghci> :i isEmpty
```

3. Zmodyfikować pierwszą linię (zapisać zmiany) pliku ex7.hs

```
module Stack (Stack, push, pop) where
```

i sprawdzić działanie

```
ghci> :l ex7.hs
ghci> :module -Stack
ghci> import Stack
ghci> :i Stack
ghci> :t MkStack
```

4. Zmodyfikować pierwszą linię (zapisać zmiany) pliku ex7.hs

```
module Stack (module Stack) where
```

wczytać plik do GHCi (potem `:module -Stack` , `import Stack`) i sprawdzić działanie

8) Moduły jako mechanizm definicji abstrakcyjnych typów danych (*ADT*)

1. W pliku ex8.hs wpisujemy

```
module Stack
  ( Stack
  , empty   -- :: Stack a
  , isEmpty -- :: Stack a -> Bool
  , push    -- :: a -> Stack a -> Stack a
  , top     -- :: Stack a -> a
  , pop     -- :: Stack a -> (a,Stack a)
  ) where

-- interface (signature, contract)
empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

-- implementation
newtype Stack a = MkStack [a] deriving Show -- hidden constructor (see the module ex

empty = MkStack []
isEmpty (MkStack s) = null s
push x (MkStack s) = MkStack (x:s)
```

```
top (MkStack s) = head s
pop (MkStack (s:ss)) = (s,MkStack ss)
```

(zapisujemy zmiany)

2. W konsoli wpisujemy

```
ghci> :l ex8.hs
ghci> :module -Stack
ghci> import Stack
ghci> empty
ghci> push 1 $ empty
ghci> pop $ push 1 $ empty
```

3. ***Zadania***:
    1. (*opcjonalne*) Zaimplementować następujący *ADT* dla kolejki

```
module Queue
  ( Queue
  , emptyQ   -- :: Queue a
  , isEmptyQ -- :: Queue a -> Bool
  , addQ     -- :: a -> Queue a -> Queue a
  , remQ     -- :: Queue a -> (a, Queue a)
  )
```

    Jaka jest złożoność czasowa poszczególnych operacji? Rozważyć możliwe optymalizacje
    2. (*opcjonalne*) Zaimplementować następujący *ADT* dla kolejki dwukierunkowej

```
module Dequeue
 ( Dequeue
 , emptyDEQ     -- :: Dequeue a
 , isEmptyDEQ   -- :: Dequeue a -> Bool
 , lengthDEQ    -- :: Dequeue a -> Int, O(1)
 , firstDEQ     -- :: Dequeue a -> Maybe a,  O(1)
 , lastDEQ      -- :: Dequeue a -> Maybe a, O(1)
 , takeFrontDEQ -- :: Int -> Dequeue a -> [a], O(n)
 , takeBackDEQ  -- :: Int -> Dequeue a -> [a], O(n)
 , pushFrontDEQ -- :: Dequeue a -> a -> Dequeue a, O(1) amortised
 , popFrontDEQ  -- :: Dequeue a -> Maybe (a, Dequeue a), O(1) amortised
 , pushBackDEQ  -- :: Dequeue a -> a -> Dequeue a, O(1) amortised
 , popBackDEQ   -- :: Dequeue a -> Maybe (a, q a), O(1) amortised
 , fromListDEQ  -- :: [a] -> Dequeue a, O(n)
 )
```

9) Organizacja kodu źródłowego: narzędzie `stack` (*ćwiczenie opcjonalne*)

1. W konsoli (terminalu) wpisujemy kolejno

```
$ stack new my-project
$ cd my-project
$ stack setup
$ tree .
```

2. Analizujemy strukturę projektu
3. (*opcjonalne*) W edytorze *Atom* otwieramy katalog `my-project`
4. Sprawdzamy zawartość plików: `app/Main.hs`, `src/Lib.hs`, `test/Spec.hs` i `Setup.hs`
5. Analizujemy zawartość plików `my-project.cabal` i `stack.yaml`
6. W konsoli wpisujemy kolejno

```
$ stack build
$ stack exec my-project-exe
$ stack test
```

7. ***Zadania***:
    1. Zapoznać się z biblioteką *QuickCheck*
    2. Zapoznać się z frameworkiem *HUnit*
    3. Napisać kilka prostych testów z wykorzystaniem obu tych narzędzi (np. dla implementacji stosu lub kolejki; można ten kod przenieść do utworzonego w tym punkcie projektu)